CS 330 – Spring 2023 – Homework 04

Due: Wednesday 3/1 at 11:59 pm on Gradescope

Note You have two weeks to work on this homework assignment. Since it is longer than the previous assignments, it will *count as 1.5 homeworks* in the final course grading.

Collaboration policy Collaboration on homework problems is permitted, you are allowed to discuss each problem with at most 3 other students currently enrolled in the class. Before working with others on a problem, you should think about it yourself for at least 45 minutes. Finding answers to problems on the Web or from other outside sources (these include anyone not enrolled in the class) is strictly forbidden.

You must write up each problem solution by yourself without assistance, even if you collaborate with others to solve the problem. You must also identify your collaborators. If you did not work with anyone, you should write "Collaborators: none." It is a violation of this policy to submit a problem solution that you cannot orally explain to an instructor or TA.

Typesetting Solutions should be typed and submitted as a PDF file on Gradescope. You may use any program you like to type your solutions. LaTeX, or "Latex", is commonly used for technical writing (`overleaf.com` is a free web-based platform for writing in Latex) since it handles math very well. Word, Google Docs, Markdown or other software are also fine.

Solution guidelines For problems that require you to provide an algorithm, you must give the following:

1. a precise description of the algorithm in English and pseudocode (*),
2. a proof of correctness,
3. an analysis of the asymptotic running time of your algorithm.

You may use anything we learned in class without further explanation. This includes using algorithms from class as subroutines, stating facts that we proved in class, e.g. correctness of subroutines, running time of subroutines and use the notation. Your description should be at the level that it is clear to a classmate who is taking the course with you.

You should be as clear and concise as possible in your write-up of solutions.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand.

(*) It is fine if the English description concentrates on the high level ideas and doesn't include all the details. But the reader should not have to figure out your solution solely based on the pseudocode. You can also add comments to your pseudocode, in fact that is best practice.

- By: Gianna Sfrisi U77992006
- Collaborators: Albert Slepak U00956163

Problem 1. (Ring road) You are responsible for redesigning the gas stations along Iceland's famous Route 1.[1] This ring road starts at mile 0 in Reykjavík, surrounds the entire island, and ends at mile 821 back in Reykjavík. Several companies have made offers on where along the road they would like to build stations, but they want to control the distance between stations to limit competition. Each offer specifies the exact location of a gas station along the road and a required minimum distance to the nearest other station.[2] For example, company A may offer to build a station at mile 23 along the road and require a minimum distance of 20 miles to the nearest other station, company B may offer to build a station at mile 348.4 and require a minimum distance of 53.2 miles to the nearest other station, etc. (Note that the locations and distances may not be integers and, since the road is a loop, a station at mile 0 may conflict with the required minimum distance of a station at mile 820 as they are only 1 mile apart.) You have received a large number of offers, some of which are incompatible, and your goal is to provide the Icelandic people with the largest number of gas stations along the road. Design an algorithm that takes as input a list of n pairs (location, minimum distance) and selects a maximum compatible subset of stations to build. *Hint:* Pick a station and find the best solution that includes this station. How can you use this information in your algorithm? *Hint:* This problem should be solved in $O(n^2)$ time.

Figure 1: Iceland's famous Route 1.

Algorithm:

/* The function isNodeValid is a helper function that uses the distances of the previous and next gas station to determine if the current gas station is valid and within the requirements*/

```
function isNodeValid(node, prev, next) {
        /* Gets the distance difference from the current node in question to the gas station before and after it*/
       distance1 = max(next.distance, node.distance);
       distance2 = max(prev.distance, node.distance);
        /* Compares both distances with the distance requirements of the gas stations involved; if it fits then the function returns true*/
        if (distance(node.position, next.position) < dist1 and distance(node.position, prev.position) < distance2)
                return true;
        else
                return false;
}
```

[1] All existing stations have been removed before you took the job, so there are currently no stations along Route 1.
[2] Only stations along Route 1 count for this purpose.

```
/* Main function of the algorithm */
largestChainLength = 0; /* Logs the length of the longest possible chain of gas stations found by the algorithm*/
largestChain = []; /* Holds the combination of gas stations associated with the longest length*/

/*Sort gas stations by the minimum required distance */
nodes = sort(nodes);

/*Iterate over every node in the list*/
for every node in nodes {
        /* Keep track of current, previous (prev), and next nodes */
        chainLength = 1; /* Iterates the number of valid gas stations */
        currentChain = []; /* Holds the set of valid gas stations*/
        for every subsequent node: {
                if (isNodeValid(node, prev, next)) {
                        chainLength++;
                        currentChain.append(node);
                }}
        if (chainLength > largestChainLength) {
                largestChain = currentChain;
        }}
return largestChain;
}
```

Run-time Analysis:
In total, the run-time of this algorithm is $O(n^2)$. The helper function isNodeValid runs at time $O(1)$ as two variables are initialized and one if statement is called. In the main function of the algorithm, there is a for loop that iterates through every node to represent starting at a different gas station each time. Nested inside the for loop is another for loop which, once the starting gas station is chosen, iterates through the remaining stations to see which gas stations can be added given their distance. These two for loops run in time $O(n^2)$ as each gas station is tested as station 0, and then compared to the other stations to create a subset of possible stations and eventually return the subset with the largest length. Additionally, since the sorting algorithm that sorts the gas stations by distance will at the most run in time $O(n^2)$, the overall run-time of the function is still $O(n^2)$.

Proof of Accuracy:
This algorithm takes into account the minimum distance per each company's requests per gas station and assesses each station according. In initializing the chain length to 0 and storing the longest chain, the function loops through every possible starting point to find the longest possible combination of gas stations given the distance requirements. Additionally, the algorithm sorts the gas station initially by the smallest distance size in order to optimize and include as many station as possible for each starting station. The for loop located in the main function of the algorithm tests every gas station as station 0, to see the impact on the total chain length by starting with a different gas station each time. The main function calls the helper function, isNodeValid, which compares the distance of the current gas station with the previous, in addition to testing the distance of the next gas station in the cycle. If including the current gas station follows both distance requirements then the helper function will return true, and the gas station will be added to the set. Once the list and chain length are determined for the current station 0, the length is compared to the previous longest chain length already discovered. The algorithm will then continue to update the largest chain variable as combinations of longer gas stations are found. The algorithm will then return the set with the largest amount of gas stations.

Problem 2. (Restaurant critic) You are a Michelin restaurant inspector and you have been tasked with thoroughly inspecting a 3-star restaurant that has recently come under criticism for declining dining standards. However, the restaurant knows that it is likely to be inspected. Therefore, your top priority is to hide your identity as a restaurant critic. Since you are in town for only one day, you want to visit the restaurant as many times as possible, but without encountering the same staff twice so as not to arouse suspicion. Your contact has provided you with a shift schedule of the n waiters of the restaurant, in which the working hours t_i of each waiter are given as intervals $t_i = [a_i, b_i)$ where a_i and b_i are times of day.[3] For technical reasons, these time intervals are considered to be closed on the left side and open on the right side. Furthermore, we assume for simplicity that the restaurant opens at time 0 and closes at time 1 so that $0 \leq a_i, b_i \leq 1$ for all $i \leq n$. Finally, we suppose that at any time in $[0, 1)$ there is at least one waiter working in the restaurant.

To complete your task, you need to find the largest set of restaurant visits[4] during the day such that each waiter works during at most one visit in that set; we call a set with this property *sparse*. Hence, you are looking for a *sparse set of visits with maximal size*. The input for this problem is the list of pairs (a_i, b_i) for $i \in \{1, \ldots, n\}$. Figure 2 shows an example of a shift schedule and orange lines indicating a sparse set. Is it the largest possible sparse set?

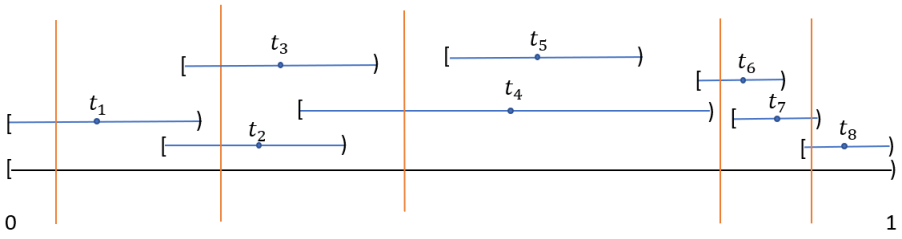

Figure 2: Example of a shift schedule. The black line represents $[0, 1)$, the operating hours of the restaurant. The blue lines indicate the working hours t_i of each waiter i. The vertical, orange lines indicate a sparse set of visits.

(a) Your friend suggests the following greedy approach: Sort the shifts by their starting times a_i. From left to right, add point a_i (the starting time of shift i) to your set of visits, as long as it does not conflict with any previous shift (i.e., no shift covers both a_i and a previously selected visit). Give an example that shows why this does not give the largest sparse set. *Note:* This algorithm chooses the starting times of the shifts, but a sparse set need not be restricted to those – any subset of $[0, 1)$ is fine.

The greedy algorithm the friend is suggesting implies that assigning visit times based on starting time of each shift will return the optimal sparse set. However, this is not always the case. As seen in Figure 2, the greedy algorithm will return a sparse set containing 4 different visits for the critic. While this greedy algorithm returns a sparse set that works with no overlap, there could be more visits based on the shifts given in Figure 2. Specifically, in regard to shifts t6, t7, and t8, the critic could add another visit time. Instead of having only one visit that intersects the shifts of t7 and t8, the critic could visit during the interval of the finish time of the t6 shift and the start time of t8. The critic would be able to increase his total amount of shifts by 1 as he could then schedule his last visit after the finish time of t7. In order to get the optimal sparse set, the algorithm needs to take the start time, finish time, and amount of overlaps into account; the algorithm cannot be restricted to just a one time type constraint.

[3] Each waiter works only *one* continuous shift.
[4] We suppose that a restaurant visit is instantaneous. Therefore, each restaurant visit is a single point in $[0, 1)$.

Before we design an algorithm for finding sparse sets, let's consider a related problem: Suppose this time you are the restaurant manager. As before, you have a fixed shift schedule by which the waiters in your restaurant work. Now suppose that, in the interest of profit, you have decided to lay off most of your staff and employ only enough waiters to ensure that at least one waiter is working at every point during the day.[5] We call such a set of shifts *sufficient*. As you aim at determining the smallest number of waiters to employ, you want to find a *sufficient set of shifts with minimal size*. How small a set of sufficient shifts can you find for the example pictured above?

(b) Prove that if there exists a sparse set $\mathcal{V}$ of visits with $|\mathcal{V}| = k$, then the manager needs at least k shifts to cover $[0, 1)$ (that is, every sufficient set $\mathcal{S}$ of shifts satisfies $|\mathcal{S}| \geq k$).

The restaurant critic cannot visit one shift more than once. Therefore if the sparse set returns a value of k, there must be at least k shifts scheduling. If there are exactly k shifts scheduled, then k is the optimal solution for both the critic and the manager; the critic was able to visit every shift and the manager was able to make a schedule with the least amount of shifts possible (very little overlap in between shifts). However, it is possible for the critic not to visit every single shift depending on the visit times the set returned, which is why there are at least k shifts and not exactly k shifts.

(c) Similarly, prove that if there exists a sufficient set $\mathcal{S}$ of shifts with $|\mathcal{S}| = k$, then the critic can find at most k visits that form a sparse set (that is, every sparse set $\mathcal{V}$ of visits satisfies $|\mathcal{V}| \leq k$).

In this example, k represents the number of shifts scheduled by the manager. The restaurant critic can only visit once per shift as they cannot visit one shift multiple times. Therefore in an optional scenario, the critic visits the restaurant k amount of times; this would equate to the critic visiting every single shift. However, depending on how optimal the manager schedules the shifts, there could be overlap in shifts which could cause the critic not to be able to visit every single shift. Hence why the critic can visit at most k shifts and it is not guaranteed the critic visits every shift once.

(d) Give a polynomial-time algorithm that takes as input a list of pairs (a_i, b_i) and that returns both a sparse set of visits *and* a sufficient set of shifts such that the two sets have the same size.

```
/* Takes in given shift_list which is the list of all the possible shifts*/
/* Sort the shifts by their starting time; if two lists have the same start time prioritize the one with
the longer duration time period */
shifts = sortByEarliestStartTimeAndDuration(shift_list);
function getSparseSet(shifts) {
        currentTimeCovered = 0;
        for (shift in shifts) {
                if (shift.endTime ≤ currentTimeCovered) {
                        shifts.remove(shift);
                } else {
                        currentTimeCovered = shift.endTime;}}
        return shifts;}
function getSufficientSet(shifts) {
        firstVisit = shifts[0].startTime;
        visitTimes = [firstVisit];
        for (shift in shifts) {
                if (shift == shifts[0]) {
                        continue;}
                visitTimes.add(shift.endTime);}
        return visitTimes; }
```

[5] You can not modify the shift schedule, and each waiter only accepts to work during their assigned shift.

Run-time Analysis:
The run-time of this algorithm is $O(n^2)$ which falls into the category of polynomial time. Both functions use a for loop and traverse through each possible shift, adding visits and deleting shifts as necessary. Since each shift and their respective start and end times are getting compared to every other shift, the overall run-time would be $O(n^2)$. Additionally, sorting the shift list at the beginning of the algorithm prior to calling the getSparseSet and getSufficientSet functions would run in at most $O(n^2)$ time. Therefore the overall run-time of the algorithm as a whole is $O(n^2)$.

Proof of Accuracy:
The algorithm consists of two separate functions; one that determines the sparse set and the second which finds the sufficient set. For both functions, the shift list is sorted by the earliest start time and then if two shifts have the same start time, the sort prioritizes the one with the longer duration. Hence the shift with the first start time and the longest duration has the first priority. The function getSparseSet keeps a log of the last time that is covered by existing shifts in the set and uses a for loop to traverse through the list of possible shifts; it compares the start time of the shifts to the current cutoff time that is already covered by existing shifts. If a shift covers a time slot that is not previously covered, it stays in the existing set and the cutoff line is updated. The function returns an accurate sparse set as it always keeps shifts required for every time slot to be covered. It removes excess shifts that occur during a previously covered time slot. Additionally, the function finds the most optimal sparse set as it prioritizes longer shifts which minimizes the total amount of shifts in the set. The second function in the algorithm gets and returns the optimal sufficient set. Since the shifts are already sorted by start time and duration, the function immediately assigns a visit at the start time of the first shift. Each additional visit is then assigned based on the end time of each of the previous shifts; this assures that no visits overlap the same shift. Since there is a priority in maximizing the number of visits, having visits assigned directly at the end time of the previous shift ensures that visits are occurring as soon as possible and with the least amount of workers scheduled at that point.

(e) Argue that the algorithm from (d) finds both a sparse set of visits of *maximal* size for the critic and a sufficient set of shifts of *minimal* size for the manager.

There is an important relationship between the sparse set and sufficient sets as a result of the parameters of this problem. Since the restaurant critic cannot visit the restaurant during the same shift twice, the critic can at most visit the restaurant k times. K represents the number of shifts the manager schedules. Therefore the sufficient set will always be greater than or equal to the sparse set. Additionally, if the two sets equal each other, this is the optimal solution for both the critic and the manager. The critic will be able to visit the maximum amount of times possible if they visit once during every single shift. And the manager would have scheduled the minimum amount of shifts to cover the entire time frame as there is very little overlap of shifts.

Problem 3. (Heaps) Design an algorithm that, given a pointer to the root of a binary min-heap and a number t, returns a list of all items in the heap that have priority less than t. Your algorithm should run in time $O(k_t)$, where k_t is the number of items with priority less than t (in particular, the running time should not depend directly on the total number of items in the min-heap).

Algorithm:

```
/*Recursive function that traverses down the heap until the parent node is greater than t */
/* takes in the left and right children of the current parent node */

    collectValidNodes(leftChild, rightChild, t) {

        {
         if leftChild != NULL and leftChild < t
                results.add(leftChild);
                /*adds node to the list of items with priority less than t */

                collectValidNodes(leftChild.leftChild, leftChild.rightChild, t);
                /* calls recursive function on the next item in the heap given the current parent item*/
        }
        {
         if rightChild != NULL and rightChild < t
                results.add(rightChild);
                /*adds node to the list of items with priority less than t */

                collectValidNodes(rightChild.leftChild, rightChild.rightChild, t);
                /* calls recursive function on the next item in the heap given the current parent item*/
        }
    return results /*returns the list of items with priority less then t */
    }
```

Run-time Analysis:
The algorithm runs in time $O(k_t)$. Each individual if statement runs in time $O(1)$. However, since the recursive function is called every time there is an item with a priority less than t, it will be called k_t times. The algorithm stops calling the recursive function once it reaches a priority greater than t therefore it will not search the remainder of the heap nor depend on the total size of the heap for its run-time.

Proof of Accuracy:
This algorithm uses recursion to trace through the binary min-heap and find the items in the heap with a priority less than the given value t. The function calls recursion on both the possible left node and right node for each parent node if both children nodes exist. The function only will continue to call the recursive function as long as there is a present node and the priority of the current root node is less than t. Once the current parent node is larger than t, the function returns the current list of items. Since this is a min-heap, it is guaranteed that all the priorities on layers beneath the current root are larger than the root node itself, which is why the algorithm stops searching once the root priority is larger than t.